Performance and Protection in the ZoFS User-space NVM File System

Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, Haibo Chen*
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Non-volatile memory (NVM) can be directly accessed in user space without going through the kernel. This encourages several recent studies on building user-space NVM file systems. However, for the sake of file system protection, none of the existing file systems grant user-space file system libraries with direct control over both metadata and data of the NVM, leaving fast NVM resources underexploited.

Based on the observation that applications tend to group files with similar access permissions within the same directory and permission changes are rare operations, this paper proposes a new abstraction called *coffer*, which is a collection of isolated NVM resources, and show its merits on building a performant and protected NVM file system in user space. The key idea is to separate NVM protection from management via coffers so that user-space libraries can take full control of NVM within a coffer while the kernel guarantees strict isolation among coffers. Based on coffers, we build an NVM file system architecture to bring the high performance of NVM to unmodified dynamically linked applications and facilitate the development of performant and flexible user-space NVM file system libraries. With an example file system called ZoFS, we show that user-space file systems built upon coffers can outperform existing NVM file systems in both benchmarks and real-world applications.

CCS Concepts • **Information systems → Phase change memory**; • **Software and its engineering → File systems management**.

Keywords non-volatile memory, user-space file systems, memory protection keys

*Mingkai Dong and Jifei Yi are supported by SOSP 2019 student scholarships from the ACM Special Interest Group in Operating Systems.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359637

ACM Reference Format:
Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, Haibo Chen. 2019. Performance and Protection in the ZoFS User-space NVM File System. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359637

1 Introduction

NVM such as phase-change memory (PCM) [32, 45], STT-MRAM [20, 30], PRAM, Memristor [50], and Intel/Micron's 3D-XPoint [46] promise to combine the best of both memory and storage. Intel Optane DC persistent memory, which was announced in 2015 [46] and has been released recently [22, 23, 25], is the first commercially available NVM product that may be widely deployed.

By combining memory's low latency, high throughput, and byte-addressability features into a high-capacity, persistent storage, NVM has attracted many research efforts on exploring new programming models [1, 57, 59], designing data structures and key/value stores [19, 21, 27, 29, 55, 60, 64], implementing file systems [5, 8, 9, 31, 33, 56, 61–63, 66], etc. NVM file systems, such as PMFS [9], NOVA [61, 62], and SoupFS [8], are proposed to leverage NVM to provide high-performance file abstractions.

Byte-addressability allows CPU to access NVM via load and store instructions. This feature has raised research interest in deploying file system libraries in user space[1] to access NVM [5, 31, 56]. However, none of the existing NVM file systems gives direct control over both data and metadata to user-space libraries. For example, file system (FS) libraries in Aerie [56] cannot directly modify the metadata stored in the file system. Instead, it needs to send expensive inter-process communications (IPCs) to a trusted process that has the authority to update metadata. Strata [31], as another example, only allows its user-space libraries to record updates in NVM logs, which requires applying the updates one more time during digestion in the kernel. Such limitations hinder fully unleashing the performance of NVM. For example, when two applications access the same file concurrently, log digestion needs to be frequently invoked, and the performance can be more than 19× slower (see §2.2).

However, several challenges prevent user-space FS libraries from gaining full control of both metadata and data. First, a

[1] We call these file systems user-space file systems although they all require kernel-side cooperation.

file system needs to enforce permissions on each file. Since the user-space FS library and the applications share the same runtime context, applications can easily bypass permission checks and conduct attacks if user-space FS libraries are given full control of the metadata. Second, applications are usually bug-prone. If user-space FS libraries are allowed to modify metadata directly, bugs in applications can easily corrupt the metadata, leaving the whole file system corrupted.

In this paper, we aim to provide direct updates in user-space file system libraries to fully exploit the NVM performance while providing sufficient protection and isolation for applications. The basic observation is that applications tend to store their files with the same permission[2] together and changes to permissions are infrequent. This indicates that it is feasible to group files with the same permission to a single container. To this end, we propose *coffer*, a new abstraction which is a collection of NVM pages that can store files with the same permission. Each coffer has a single permission shared by all pages within it.

With coffers, we build an NVM file system architecture, *Treasury*, to separate NVM protection from management so that user-space libraries can manage NVM within a coffer with *full control* and cross-coffer protection is guaranteed by the kernel. The kernel part of Treasury records the metadata of all coffers and is responsible for checking permissions for coffer-level requests from the user-space file system libraries. User-space libraries are responsible for managing file system structures within the coffers, including the modification of metadata. NVM resources are mapped to user space at coffer granularity, which avoids mapping the whole NVM to processes such as that in Strata [31].

Treasury relaxes the protection granularity from a file to a coffer. Although it retains protection under normal execution, Treasury may still be subject to corruption by application bugs. Besides the paging mechanism enforced by the hardware memory management unit (MMU), Treasury leverages Intel memory protection keys (MPK) to efficiently prevent application bugs from corrupting the file system data and metadata and prevent corruptions from being spread across processes and coffers.

Treasury provides general file system functionalities like user-space file descriptor (FD) mapping tables and symbolic links so that unmodified real-world applications can run on Treasury without recompilation. We implement a synchronous file system called ZoFS using Treasury and evaluate ZoFS with file system benchmarks and real-world applications. The evaluation results show that ZoFS outperforms existing NVM file systems in most cases for both FxMark and Filebench. For real-world applications, ZoFS reduces LevelDB latency by up to 82% and improves SQLite throughput by up to 31%.

[2]For brevity, we use the word "permission" to indicate a combination of the read-write permission and the owner user and group in the paper.

Table 1. DRAM and Optane DC PM latency and bandwidth.

Memory	Operation	Bandwidth	Latency
DRAM	read	115 GB/s	81 ns
	write	79 GB/s	86 ns
Optane DC PM	read	39 GB/s	305 ns
	write	14 GB/s	94 ns

Table 2. Latency (ns) of operations on a file/directory shared by multiple processes.

Operation	# Processes	Strata	NOVA	ZoFS
append	1	1,653	2,172	1,147
	2	34,551	3,882	1,703
create	1	4,195	3,534	2,494
	2	283,972	6,167	3,459

In summary, the contributions of this paper include:

- The insight on file permissions and permission operations through typical file system traces (§2);
- A new abstraction, *coffer*, and a user-space NVM file system architecture called Treasury that allows direct management over NVM resources in user space while providing protection and isolation (§3 and §4);
- An example file system, ZoFS, using Treasury (§5);
- A detailed comparative evaluation of ZoFS and existing NVM file systems on Intel Optane DC PM (§6).

2 Background and Motivation

2.1 NVM and NVMFS

The recent release of Intel Optane DC persistent memory [46] marks the transition of NVM from concept exploration to large-scale commercial deployment [22, 23]. Table 1 shows the basic performance characteristics of Intel Optane DC persistent memory, compared with DDR4 DRAM [25]. Beyond the significant speed improvement over traditional storage such as HDD or SSD, NVM can be directly attached to the memory bus and be accessed at byte-granularity via CPU load/store instructions, giving much more potential of its usage, blurring the boundary of storage and runtime memory.

The promising characteristics of NVM have encouraged designs of numerous NVM file systems. Ext4-DAX [4, 6, 58] and XFS-DAX [3] are two examples of mature file systems modified to support NVM with page cache bypassing. BPFS [5] is a B-tree NVM file system optimized via short-circuit shadow paging. PMFS [9] is a journal-based file system designed for NVM. NOVA [61, 62] brings the idea of log-structured file systems on NVM, and SoupFS [8] revisits the soft updates technology on NVM to build an NVMFS that is simple and fast.

2.2 User-space NVMFS and the deficiency

NVM's byte-addressability also inspires researchers to design NVM file systems in user space.

Aerie [56] is a flexible file system architecture for NVM applications. The goal of Aerie is to expose NVM to user-space applications so that they can access file data without

Table 3. File permissions in databases and web servers.

System	Type	Perm.	Uid/Gid	# Files	Size
MySQL	Directory	750	970/970	6	32KB
	Regular	640	970/970	358	399MB
	Regular	644	0/0	1	0B
PostgreSQL	Directory	700	969/969	28	128KB
	Regular	600	969/969	1,807	99MB
DokuWiki	Directory	755	33/33	1,035	5MB
	Regular	644	33/33	19,941	452MB

the interaction with the kernel. FS libraries in Aerie can read the metadata stored in NVM to find files, and then they can read and modify the file data directly. However, when FS libraries want to update metadata, they need to send requests to a trusted file system service via IPCs, which are expensive.

Strata [31] is a cross-media file system designed for multiple layers of storage media. All reads to data and metadata are directly processed in user space. For updates, Strata uses a separate NVM device as a log device and logs all updates in user space. The logs are gradually digested to lower layers, which can be NVM, SSD, or HDD. The log-then-digest updates cause the double-write problem and are inefficient when multiple processes share the same file.

Since user-space NVM file systems can perform most reads in user space, they reduce the number of system calls and usually run faster than the kernel-space counterparts. However, existing user-space file systems also impose limitations on user-space libraries: direct updates, especially direct metadata updates, are strictly forbidden.

We illustrate the impact of indirect updates by measuring the average latency of appending 4KB data in a shared file and creating an empty file in a shared directory. The results are shown in Table 2, where we can observe the `append` and `create` latency of Strata is considerably higher when two processes share the same file or directory. For reference, we also conducted the same test on NOVA and the file system (ZoFS) that we will introduce in §5. Strata's `append` performance is better than NOVA, while `create` is relatively slower because Strata has to write two logs for each `create` to ensure the metadata consistency.

The deficiency of indirect updates motivates us to think: how can we design a user-space FS that can have full control over both data and metadata on NVM to exploit NVM performance while guaranteeing sufficient protection and isolation?

2.3 Permissions and isolation

File systems use permissions to restrict specific files an application can access, and this practice protects data stored in the file systems. To enforce the permissions, file systems are isolated from applications and provide services via limited interfaces such as system calls. To exploit NVM performance by giving user-space file systems full control over both data and metadata, we need to reconsider the isolation between applications and file systems. Thus, we conducted a survey on file permissions of application data.

First, we surveyed file permissions in the data directory of two databases, MySQL and PostgreSQL, and a long-running DokuWiki website[3]. For each database system, we initialized a new data directory and imported example databases (employee, world, and sakila for MySQL [39]; and World, dellstore2, and Pagila for PostgreSQL [44]). DokuWiki uses files to store information about our laboratories, such as projects, members, and publications, so we directly use these files for analysis. Table 3 shows the results. For both database systems, the permissions of files are highly concentrated to `640` and `600` for regular files. The only exception for MySQL is an empty "debian-5.7.flag" file owned by the root user with `644` permission, which indicates the database binary format version for upgrade [12]. For our DokuWiki website, all regular files have `644` permissions. We also found that permissions of these database files are never changed during the process of database requests. As for DokuWiki, the only operation that may change the file permission is a `chmod` system call upon file uploads. But this `chmod` operation only changes the file permission when the permission does not match DokuWiki's PHP mask, which never happens in our setup. These findings show that *applications tend to store files with similar permissions that are rarely changed*. Thus, it seems feasible to group different files with the same permissions to reduce context switches for permission checks and give user-space FS libraries more flexibility when managing these files.

To confirm our findings, we further analyzed FSL Homes Traces [47], which are a collection of daily snapshots of students' home directories from a shared network file system [52]. We chose to analyze the latest snapshot, which is taken on April 10th, 2015 and includes 15 home directories. The summary is given in Table 4, where we only show file statistics aggregated by permission bits due to space limitations. There are 726,751 files in total, and 89% of them are regular files. `644` is the most popular permission for regular files, followed by `600`. Since no detailed information about directories is given in the trace, we assumed the directory ownership and permission to be the same as the first file we analyzed in the trace. We ignored the execution bit in file permissions since it is hard to restrict such permissions after mapping NVM to user space.

We attempted to group files with the same permission and owners by the following rules. If a file has the same permission as its parent, then it stays in the same group as its parent. Otherwise, a new group is created, and the file is put into the new group. We started from a single group containing the FS root directory and grouped files top-down. As a result, 4,449 groups are formed, with the largest group containing about 1/3 of all files. We also compute the min/average/max sizes of file groups for different permissions. The largest group contains files of 52.0 GB while the average group size is about 79.7 MB. This proves that if we give full control of one group

[3] https://ipads.se.sjtu.edu.cn

to user-space FS libraries, the user-space FS libraries can manage files in the group internally without suffering from the context switches to/from the kernel. It is worth noting that there are also 3,795 single-file groups. However, the sum of file numbers in these groups only takes 0.6% of the total file number.

We investigated further for permission changes during application executions since FSL Homes Traces only contain the static state of a file system. MobiGen Traces [26, 48] contain 2 minutes worth of I/O system call traces collected on a Samsung Galaxy smartphone, and two traces are given. In the Facebook trace, we found no `chmod` or `chown` system calls among 64,282 system calls. In the Twitter trace, there are 16 `chmod` and no `chown` system calls in 25,306 system calls. The 16 `chmod` system calls are used regularly in a fixed pattern, where a shadow file is created with `600`, written with new data, and then changed to `660` before being renamed to replace the actual file.

The survey results support our findings and encourage us to group files with the same permissions and give user-space FS libraries full control when managing these files.

2.4 Memory protection keys

Intel memory protection keys (MPK) [7, 24, 40] is a new hardware feature that allows user-space applications to restrict their memory accesses. MPK allows the kernel to store a four-bit region number in each page table entry. Thus, the memory space of a process can be separated into at most 16 regions.

In addition, MPK provides a CPU register called `PKRU`, which contains 16 pairs of two-bit permissions indicating whether each region is read-only, read-write, or inaccessible. Each thread has its own `PKRU` register value, and each thread can change its accessibility to each region by manipulating its `PKRU` via a non-privileged instruction `WRPKRU`.

On each memory access, the MMU will read the region number in the page table entry and check the corresponding permissions in the `PKRU` register. If the memory access violates the permission, a page fault will be triggered and delivered to the user-space application. MPK is supplementary to the existing page permission bits in the page table entry, and both permissions will be checked during memory access.

In summary, MPK allows the kernel to divide the memory space of a process to multiple regions and allows each thread to restrict its own access to each region by writing to a non-privileged CPU register.

3 Design

Given the observation that we can group files with the same permission, it is possible to give user-space FS libraries full control to manage these files if they have the permission. In this section, we first introduce the coffer abstraction (§3.1), which can contain a group of files with the same permission. Then we use coffers to design the Treasury architecture (§3.2) and show how user-space FS libraries communicate with the kernel part using coffer interfaces (§3.3). We show how protection and isolation can be further enforced (§3.4) and then discuss crash recovery (§3.5).

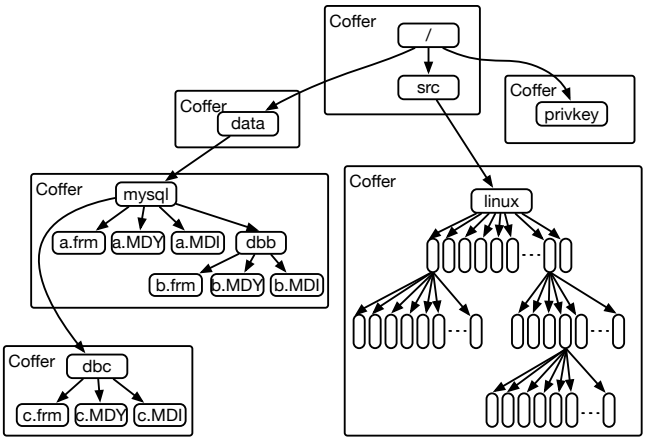


Figure 1. An example of using coffers in the file system hierarchy. Each coffer contains a single regular file or a directory with some of its children.

3.1 The coffer abstraction

We introduce *coffer*, a collection of NVM pages that share the same permission, to store files. Each coffer owns a *root page*, which contains metadata about the coffer. The root page is managed by the kernel and can be mapped as read-only to user-space libraries.

Figure 1 shows an example of using coffers in the file system hierarchy. In the example, besides a root page, each coffer has a root file, which is the entry of the coffer. If the root file is a file other than a directory (e.g., a regular file or a symbolic link), the coffer stores only the root file. If the root file is a directory, its child files can be stored within the same coffer, so that they can be managed together by user-space FS libraries without the involvement of kernel. When the child files are not stored in the same coffer of the parent, cross-coffer reference is recorded in the dentry so that user-space FS libraries know in which coffer they will find the target file.

The coffer abstraction is the key enabler for direct management of files in user-space FS libraries. Since files in a coffer have the same permission, we only need to check the permission when the coffer is first requested by a process. Once the access is granted to the process, all NVM pages in the coffer are mapped to the process so that FS libraries in the process can read/write these pages directly. By dividing NVM spaces into multiple coffers, the permission of the file system can be enforced at coffer granularity, and FS libraries can access data and metadata in coffers directly as long as they are mapped.

3.2 Treasury

Based on coffers, we build Treasury, a user-space NVM FS architecture, to fully exploit the performance advantages of NVM while providing sufficient protection and isolation.

Table 4. File statistics in an FSL Homes Trace snapshot. The left side categorizes files according to the type. The right side breakdowns the number of files according to the permission. The bottom rows show the number of groups we classify files into and the group sizes.

Type	# Files	644	600	666	444	660	640	664	440
Regular	648,691	538,538	105,226	233	3,313	342	921	110	8
Symlink	6,486	18	0	6,468	0	0	0	0	0
Directory	71,574	65,127	4,021	927	1,099	276	33	91	0
All Files	726,751	603,683	109,247	7,628	4,412	618	954	201	8
# Groups	4,449	1,935	1,174	365	48	15	853	51	8
Min Size	0B	0B	0B	7B	660B	23.5KB	0B	28.7KB	455B
Avg Size	79.7MB	46.1MB	222.2MB	474.2KB	92.5MB	118.2KB	31.9KB	348.2KB	26.5KB
Max Size	52.0GB	23.4GB	52.0GB	106.7MB	995.1MB	211.1KB	10.5MB	5.4MB	98.3KB

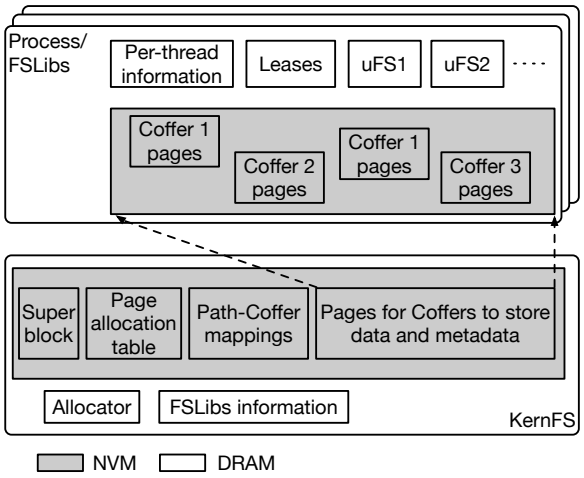


Figure 2. Treasury consists of a kernel module (KernFS) and a user-space library (FSLibs). KernFS maintains page allocation information and path-coffer mappings in the file system. FSLibs contains a collection of FS libraries (μFSs) and some auxiliary tools. The internal structure of a coffer is maintained by a corresponding μFS. A μFS to a coffer is similar to a file system to a device partition.

Figure 2 shows the architecture of Treasury. Treasury is comprised of two components: a kernel module (KernFS) in kernel space and a library (FSLibs) in user space.

KernFS is responsible for global space management. It uses an allocation table to record whether a page on NVM is allocated to a coffer or not. It also manages all coffers in a path-coffer mapping table. KernFS treats coffers as black boxes. It only knows metadata about a coffer, such as the coffer path, the coffer type, and which pages belong to the coffer. It does not need to know what is stored in the coffer and how the data is organized.

FSLibs, on the other hand, is a collection of FS libraries, which we call μFSs, and tools that help μFSs manage data and structures within coffers and interact with applications. There can be multiple μFSs in FSLibs, and each of the μFSs can manage a certain type of coffer. A μFS to a coffer is similar to a file system to a device partition. Different μFSs can take different approaches to organizing file data and metadata in the coffer. Different types of coffers are distinguished by the coffer type in the coffer metadata.

3.3 Interfaces

A μFS needs to communicate with KernFS for some coffer-level requests. Communication is done via the `ioctl` system calls in the implementation. Requests are classified into three categories, as listed in Table 5.

Coffer operations. Coffer operations are used to modify the coffer metadata recorded in KernFS. When a μFS needs to create or delete a coffer, it explicitly sends `coffer_new` or `coffer_delete` requests to KernFS who will check the validity and process the request. The same procedure also goes for `coffer_enlarge`, which requests free NVM pages in batch from KernFS, and `coffer_shrink`, which releases free pages back to KernFS. On each `coffer_map` operation, KernFS will first check whether the process has the permissions to access the coffer. Once the check passes, KernFS will map the corresponding pages to the process's memory space, so that a μFS can access the coffer directly in user space. The mapping is removed when the `coffer_unmap` is requested or when the process's user and group identifiers are changed by system calls such as `setuid`. A `coffer_split` operation splits a coffer to two coffers, which is used to modify the permission of some part of the original coffer. A `coffer_merge` operation does the opposite; it merges two coffers into a single one. The `coffer_recover` operation is invoked during the recovery of a coffer, which will be discussed in §3.5. By using these coffer operations, μFSs can manage the coffer-level metadata under the supervision of KernFS.

FS operations. Two operations are used by FSLibs to register or deregister itself from KernFS. In an `fs_mount` operation, KernFS allocates structures to track the information for the new FSLibs instance. These structures are released in an `fs_umount` operation or when the process is terminated.

File operations. Two operations, `mmap` and `execve`, are treated specially in Treasury because they cannot be done in user space. Both calls require a knowledge of file internal structures and need higher privileges to modify the page table for the process. By introducing `file_mmap` and `file_execve`

Table 5. Protocols between KernFS and FSLibs. These operations are requested by μFSs in FSLibs and validated and processed by KernFS.

Operation	Description
coffer_new	Create a new coffer under the given coffer.
coffer_delete	Delete an existing coffer.
coffer_enlarge	Allocate and assign more pages to a coffer.
coffer_shrink	Free some pages from the coffer.
coffer_map	Map a coffer into the process' memory space.
coffer_unmap	Unmap a mapped coffer.
coffer_split	Split a coffer into two.
coffer_merge	Merge two coffers into a single one.
coffer_recover	Recover a coffer.
fs_mount	Register a new FSLibs instance.
fs_umount	Deregister the FSLibs instance.
file_mmap	Map a file.
file_execve	Execute a file.

interfaces, μFSs can provide the data locations to KernFS, so that KernFS can update the page table with the data addresses.

3.4 Isolation and protection

Treasury relaxes the isolation boundary from a file to a coffer and allows the user-space FS libraries to manage a coffer directly once it is mapped. However, user-space direct management also incurs several issues in isolation and protection, which we address in this subsection.

3.4.1 Stray writes

Stray writes [9] happen when the control flow is messed up because of bugs. Since NVM can be modified by simple CPU store instructions, stray writes can easily corrupt existing data and structures stored on NVM. In prior work such as PMFS [9], stray writes are protected using a combination of page table isolation and the CR0.WP bit. Buggy processes cannot modify the state of NVM mapped in other processes or the kernel because of the isolation provided by the page table and the privileged mode. On the other hand, buggy writes in other parts of the kernel (e.g., drivers) are prevented from modifying NVM state by a write window mechanism. When NVM is read-only mapped in kernel space, the NVM is only modifiable when the CR0.WP bit is cleared. Thus, PMFS clears the CR0.WP before it modifies NVM and sets the bit right after the modification. This temporarily opens a write window in which the NVM can be modified while leaving NVM read-only most of the time.

Stray write protection is much harder for Treasury because NVM coffers can be directly modified in user space. One straightforward solution is to map the coffer as read-only and makes it writable only when a μFS wants to update the NVM. However, this solution requires lots of page table permission updates and cannot prevent stray writes by other concurrent threads in the same process.

Protection from stray writes. In Treasury, we extend the idea of write windows from PMFS by using Intel MPK to protect file systems from stray writes.

When KernFS maps a coffer to a process, KernFS will mark the coffer pages in a different MPK region from the process's runtime memory. KernFS will also disable access to the coffer's MPK region in the thread's `PKRU` register, before returning to user space. To protect NVM from stray writes, μFSs should obey the following guideline:

G1. A coffer can be accessible only when the μFS is accessing the coffer.

When a μFS wants to access a coffer, it first enables the access permission to the coffer's region by updating the `PKRU` register. This permission is disabled right after μFS finishes the access. As a result, no coffers are accessible when the application code is running, and since the μFS code is trusted, no stray writes can corrupt coffers.

This protection is efficient because updating the `PKRU` register requires a single `WRPKRU` instruction, which has little overhead (about 16 cycles on our platform). Since the register is per-thread, stray writes in other concurrent threads cannot leverage the access window opened by the normal thread, which further enforces the integrity of the protection.

3.4.2 Graceful error return and fault isolation

Although software bugs in applications can be effectively prevented from corrupting coffers, there can still be data and metadata corruptions caused by malicious attacks or hardware faults. Data corruptions can do limited harm to a running process, since user-space FS libraries simply return the wrong data. Metadata corruptions, however, can either cause abnormal termination of the whole process or lead user-space FS libraries to spread the corruption to other coffers that the process has mapped. For such situations, we again leverage Intel MPK to minimize the interference to normal processes and isolate the fault within a coffer.

Graceful error return. Kernel file systems return an error code when there is something wrong during the system call. However, if some coffer is corrupted while FSLibs is accessing the coffer, FSLibs is likely to access invalid memory areas, which will cause the whole process to be terminated by segmentation faults.

A feasible solution to the problem is to check the validity of the address before each access in FSLibs. But this solution requires too many address checks on the critical path.

We, instead, hook the segmentation fault handler, so that the handler could translate the cause of the fault into a file system error and report it to the application. Specifically, we invoke `sigsetjump` at the beginning of FSLibs file system functions and call `siglongjump` in the `SIGSEGV` signal handler to jump back to the beginning of the FSLibs function and return corresponding errors. This approach protects the application from being terminated due to invalid memory references in a coffer. As a result, errors in FSLibs can always

be converted into a file system error and gracefully returned to the application.

Fault isolation. To prevent corruptions in one coffer from spreading to other coffers, KernFS maps different coffers to different MPK regions. A μFS should obey the following guideline.

G2. At any time, at most one coffer is accessible in user space for each thread.

A μFS should keep track of which MPK group each coffer belongs to and enable the access permission only to that region before access. As a result, metadata and data accesses in unintended coffers will be prevented by MPK and faults cannot propagate across coffers.

Note that even if only one coffer is accessible, there can be multiple coffers mapped in the process, and switching the accessible coffer among the mapped coffers only requires a `WRPKRU` instruction. There are at most 15 coffers simultaneously mapped to a process, since only 15 MPK regions are available. If a `coffer_map` operation is requested and KernFS finds no more available regions, KernFS will return an error, and the μFS should call `coffer_unmap` to release MPK regions before mapping new coffers.

3.4.3 Metadata security

In traditional file systems, a malicious process can only attack other processes by modifying a file shared by other processes. In Treasury, however, since metadata can be directly modified in user space, a malicious process may also attack others by manipulating the metadata in shared coffers.

If the manipulated metadata does not involve other coffers, the victim may read the wrong data or receive file system errors for corrupted coffer structures. For the former situation, the attacker can also achieve the same effect via normal FS operations since it has permission to write all files within the coffer. For the latter situation, the victim will neither leak data nor do wrong modifications.

For manipulated metadata involving other coffers, Treasury prevents the effect of manipulated metadata from spreading to other coffers to defend from such metadata attacks. Assume two processes share the same coffers, and the malicious process (the attacker) tries to compromise the other (the victim) by manipulating shared metadata in coffer A. Supposing the victim obeys *G1* and *G2*, when the victim accesses coffer A, it will make coffer A accessible and all other coffers inaccessible. Thus, if the victim follows the manipulated metadata and accesses other unexpected coffers, MPK violations are triggered, and the victim can stop its file access.

On the other hand, when the victim meets an expected cross-coffer reference, it will proactively disable the current coffer's accessibility and make the target coffer accessible. In case an attacker manipulates a cross-coffer reference, we have the last guideline for μFSs to enforce the integrity of cross-coffer references:

G3. For each cross-coffer reference, μFSs should check the validity of the target coffer before making it accessible.

In the example μFS ZoFS in §5, the dentry is the only structure that may contain cross-coffer references, as it may point to the root inode of another coffer. The only chance for the attacker to bypass MPK protection and propagate manipulated metadata to other coffers is to modify these cross-coffer references in coffer A's dentries. However, when ZoFS in the victim process accesses a dentry containing cross-coffer references, it will check the source dentry's path against the target coffer's path to ensure that the coffer to be accessed is indeed expected. The cross-coffer reference is also validated to ensure that it points to the root inode of the target coffer. As a result, no matter how the attacker manipulates the metadata within a coffer, the victim, who shares the manipulated coffer, can prevent manipulated metadata from being spread to other coffers.

Metadata in coffers can be manipulated and become invalid. Thus, if applications directly read and use metadata in their own way, they should check the metadata to prevent attacks such as buffer overflows. Note that all coffers are mapped to user space as non-executable; thus, Treasury does not facilitate attacks that inject malicious code.

3.5 Recovery

A coffer needs recovery when it is corrupted. The recovery can be initiated by any μFS. The μFS, which is the initiator, invokes `coffer_recover` to notify KernFS the recovery request. KernFS marks the coffer as "in-recovery" with a recovery lease in the coffer root page, and then unmaps the coffer from all processes other than the initiator, and returns to the initiator. The initiator starts recovery within the coffer and sends the addresses of in-use pages to KernFS, who will compare them to pages allocated to the coffer and reclaim pages that are not used.

4 Implementation

In this section, we focus on the issues and solutions in the implementation of Treasury to support modern real-world applications.

4.1 KernFS

Coffer management. KernFS maintains all coffers information in the whole file system. As defined in §3.1, each coffer has a root page that stores the metadata of the coffer. Treasury uses the relative address of the root page (i.e., the `coffer-ID`) to identify each coffer. Treasury also introduces a persistent hash table (the path-coffer mappings in Figure 2) to store all coffers. The key of the hash table is the path of the coffer, and the value is the `coffer-ID`. We use paths to index all coffers to boost file lookup. When a μFS wants to access a file, KernFS can find and map the coffer containing the file by comparing the file path and paths stored in the hash table. Coffers can also be directly located by calculating the address using the `coffer-ID`.

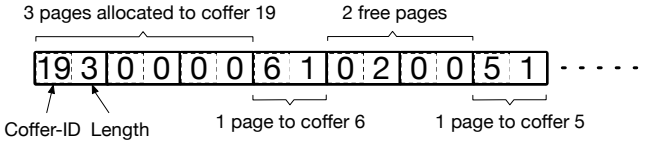


Figure 3. The allocation table used in KernFS to record per-page allocation information. For each page, the left-side integer means whether the page is free (with 0) or allocated to some coffer (with coffer-ID). The right-side integer indicates how many consecutive slots, including the current one, share the same coffer-ID.

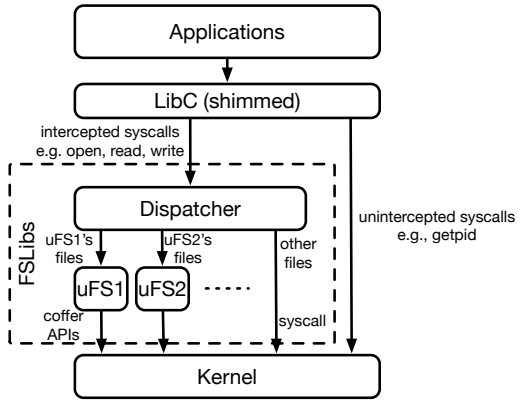


Figure 4. The architecture of FSLibs.

Space management. Treasury adopts two-level allocations: KernFS allocates NVM pages in batch to coffers, and each coffer further allocates its pages to store data and metadata.

KernFS manages all NVM space globally at page granularity. It leverages a persistent allocation table (as shown in Figure 3) to track the allocation state of each page. In the allocation table, each page's allocation state is indicated by a 32-bit integer (the `coffer-ID`), which records to which coffer the page is allocated. A zero `coffer-ID` indicates free pages, i.e., pages that have not been allocated to any coffer.

To speed up allocation of a large number of consecutive pages, adjacent allocation slots with the same `coffer-ID` are merged. As a result, a new 32-bit integer is added for each page, indicating how many consecutive pages, starting from this page, share the same `coffer-ID`.

We also use some volatile data structures to track and boost allocation. For example, we use a global volatile red-black tree to track all free space in the allocation table, and another red-black tree to track all allocated space and the corresponding `coffer-ID`.

4.2 FSLibs

The goal of FSLibs is to transparently run dynamically linked applications on user-space NVM FS libraries. By preloading FSLibs, the applications can access files managed by user-space NVM FS libraries (i.e., μFSs) without recompilation. Figure 4 shows the architecture of the FSLibs in Treasury, where there is a dispatcher and one or multiple μFSs.

System call interception. We add shim functions in glibc to intercept all file system related system calls.

FSLibs is compiled to a dynamic library named `libfs.so` and loaded by the `LD_PRELOAD` environment variable during the boot of applications. Afterwards, file system related system calls invoked by the applications are redirected to the dispatcher, which will dispatch the system calls to the corresponding μFS according to the coffer type.

We need to distinguish files in FSLibs and files stored in kernel file systems. For a filename, if it is an absolute path, we compare it with FSLibs's mount path, to see whether the file is stored in FSLibs. For a relative path, we prepend the maintained current working directory path to the relative path before comparing it to the FSLibs mount path. For file descriptors (i.e., FD), we maintain a user-space FD mapping table and look up the table each time when we get an FD.

FD mapping table. Prior work [31] differentiates FDs of user-space FS libraries and kernel file systems by comparing them with a threshold. For example, FDs larger than 5000 are reserved for user-space FS libraries, and the rest are used by kernel file systems. This is simple, but it causes problems when applications, such as bash, depend on system calls such as `dup`. `dup` duplicates an existing FD and returns the new FD. Additionally, the returned FD should be the lowest available FD number [35], which cannot be satisfied by separating the range of user-space and kernel FDs.

To solve the problem, we choose to maintain a user-space FD mapping table, in which FDs used by applications are mapped to FDs used by kernel or file structures in μFSs.

As a result, each system call that involves FDs is intercepted, and FDs are translated by the user-space FD mapping table. With the table, we can make sure the `dup` call will always return the lowest available FD the application can observe, which satisfies the requirement of applications.

As a side effect, we need to maintain the user-space FD mapping table across system calls such as `exec`, `clone` and `vfork`. For example, we serialize the FD mapping table content using base64 and pass it across `exec` calls using a dedicated environment variable.

Symbolic links. Implementing the symbolic link (i.e., symlink) file is easy, but following symlinks is complex during the path walk. We use a clumsy method to handle symlinks in the page walk: whenever one symlink is expanded in a μFS, the new path will be returned to the dispatcher, which will re-dispatch the file request. Such an implementation can correctly handle most of the symlink cases. One exception is a symlink in kernel file systems that points to FSLibs files, which we will consider as future work.

4.3 Limitation and discussion

Although Treasury tries to provide the same functionality as kernel file systems, there are limitations that Treasury cannot bypass easily.

Permissions. Treasury takes page permissions enforced by the page table to restrict whether a process can read or write a coffer. However, each page table entry stores a single bit to

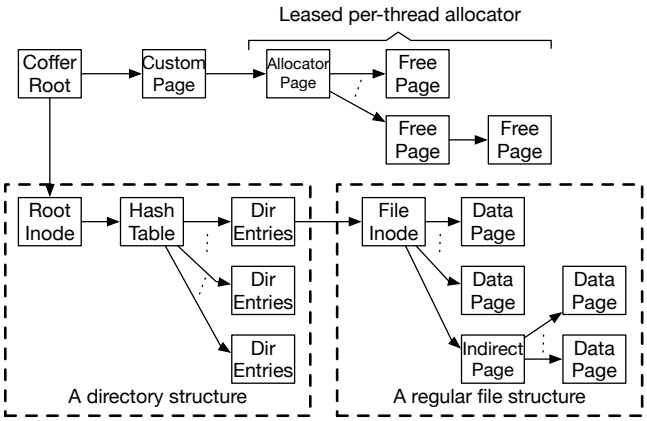


Figure 5. Overview of ZoFS. Each block represents a page. The coffer root page is required by Treasury, while the rest of the structures are designed by ZoFS.

indicate the page is read-only or read-write; thus, file permissions such as write-only cannot be easily implemented. For the execution permission of a file, Treasury always maps the coffer pages as non-executable; and the execution permission is maintained by μFSs without kernel enforcement. Treasury also cannot support complex permission mechanisms such as POSIX access control lists (ACLs).

Permissions of directories have different meanings from those of regular files. For simplicity and efficiency, Treasury ignores the difference and treats directory permissions as inaccessible, read-only, and read-write, which is the same as regular files.

Access time. Treasury cannot support `atime` easily for read-only files since the `atime` metadata cannot be updated in user space for read-only coffers.

Denial-of-service attacks. In Treasury, FSLibs are given full control over data and metadata with mapped coffers, which makes FSLibs capable of imposing denial-of-service (DoS) attacks such as holding a lease lock and never releasing it.

Applications using MPK. Treasury heavily relies on the MPK mechanism. If the applications also need to use MPK, the applications and Treasury will compete for the limited MPK regions in the same virtual memory space.

5 ZoFS: An example user-space NVM FS

The internal structures of coffers are defined by a specific μFS implementation. In this section, we introduce an example μFS called ZoFS that uses coffers and is implemented in FSLibs.

As described in §3.1, ZoFS manages files in a tree hierarchy (Figure 1): a file can be stored in its parent's coffer only when it has the same permission as its parent. This is a design decision of ZoFS, and other μFSs can choose different approaches. For example, a μFS can use a flat hierarchy to organize all descendant files of a directory. It can divide these files in different coffers only by their permissions, ignoring their hierarchical relations.

Figure 5 shows the architecture of a ZoFS coffer. Each coffer is allocated by KernFS with three pages. The first page is used as the coffer root page, which is the entrance of the coffer and contains metadata about the coffer, such as the coffer path and type. The other two pages are a page for root file inode and a custom page for per-coffer data. The addresses of these two pages are stored in the coffer root page, which is read-only for ZoFS.

5.1 Data and metadata organization

ZoFS only supports 4KB-sized allocation for simplicity, so structures used to organize files and directories are 4KB-aligned. Techniques, such as embedding file data in the inode page, can be used to improve the space efficiency, which we leave as future work.

Directories. ZoFS uses adaptive two-level hash tables to organize directory entries. The first-level hash table has 512 pointers, each of which points to a second-level page. The second-level page consists of two parts. The first half of the page stores some directory entries (e.g., dentries) and the second half stores a second-level hash table with 256 buckets. Each bucket of the second-level hash table stores a linked list of pages containing dentries. ZoFS tries to put new dentries in the second-level page first, and only when the second-level page is full, the dentries are inserted into the second-level hash tables. Pages in a directory are allocated on demand to reduce unnecessary storage overhead.

Each dentry contains a hash value of the filename for fast checks, the filename itself, the `coffer-ID` and the inode pointer. A zero-value `coffer-ID` indicates the inode is within the same coffer as the directory. Accessing an inode with non-zero `coffer-ID` requires switching the MPK region or mapping a new coffer via `coffer_map`.

Regular Files. ZoFS manages regular files in a way similar to Ext4. The file inode contains pointers to data pages, indirect pages, and double indirect pages.

Special Files. An inode in ZoFS consumes a 4KB page. Thus there is sufficient space to store data of special files, for example, the target of a symbolic link.

5.2 Leases and allocation

Multiple threads, both intra- and inter-process, can manage the same data structures in a coffer at the same time. ZoFS uses lease locks in case some processes are terminated unexpectedly while holding the locks.

Lease locks require globally synchronized timestamps across all CPUs. ZoFS uses the `clock_gettime` call to get timestamps for leases. Thanks to the virtual dynamic shared object (vDSO) feature [36], `clock_gettime` is efficient since it can be processed in user space without invoking system calls to the kernel.

Lease locks can protect concurrent accesses to shared objects. However, it does not help scalability on high-contention structures such as allocators. In ZoFS, we combine lease locks

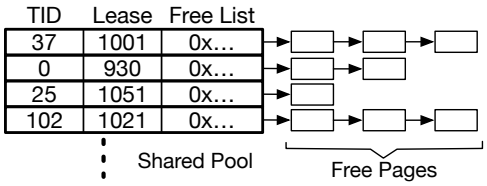


Figure 6. Leased per-thread allocator.

and thread-local storage (i.e., per-thread variables [13]) to improve the scalability of allocators. Figure 6 shows the leased per-thread free list in allocators. The leased per-thread free list structure contains a thread ID (TID), a lease and the free list header.

ZoFS pre-allocates sufficient leased per-thread free list structures in a shared pool. When a thread wants to allocate NVM pages, it first checks whether it has already held a leased per-thread free list structure[4] and whether the lease is still valid. If both are true, it renews the lease and directly allocates from the free list stored in the structure. Otherwise, it needs to find a free leased per-thread free list structure from the shared pool and update the TID and lease information before using the free list. If the free pages in one free list are insufficient, ZoFS sends a `coffer_enlarge` request to the kernel to acquire more NVM pages.

The leased per-thread allocator improves the scalability of ZoFS's allocation. Also, even if a thread is terminated unexpectedly, its per-thread free list can be reused by others when the lease expires.

5.3 Consistency and recovery

To ensure consistency, ZoFS issues atomic instructions with proactive cache line flushes to ensure atomic metadata updates. Each metadata update is divided into multiple steps. ZoFS follows a carefully-decided order of these steps, so that after crashes, partially processed metadata updates can be rollbacked by the recovery code. This is similar to the approach used in SoupFS [8]. ZoFS does not implement atomic data updates for simplicity.

ZoFS's consistency relies on the recovery phase. Although Treasury provides an approach for online recovery, ZoFS only implements an offline recovery.

For recovery, ZoFS first scans all coffers in the whole file system and recovers in-coffer metadata for each coffer as follows. It starts with the coffer root page to traverse the whole coffer and records in-use NVM pages and cross-coffer metadata. When a corrupted file or dentry is found during the traversal, ZoFS first tries to recognize and recover it. If the recovery is not possible, ZoFS skips the corrupted content. At the end of traversal, ZoFS sends all in-use pages in the coffer to KernFS, which will reclaim the rest of the pages. After all in-coffer metadata are checked, ZoFS continues to validate cross-coffer metadata according to the information recorded during coffer traversals.

[4]The information is recorded in a normal per-thread variable.

ZoFS is a simple example of a μFS in Treasury. It is possible to implement other kinds of μFSs in Treasury with the coffer abstraction. For example, one can implement a journaled μFS or a log-structured μFS in Treasury as well.

6 Evaluation

We present evaluation results for ZoFS in this section. To present the characteristics of ZoFS comprehensively, we first evaluate and analyze the basic performance of ZoFS using a set of micro-benchmarks. Then we use macro-benchmarks to show how ZoFS performs under synthesized workloads, and finally run real-world applications on ZoFS to reveal its advantages and disadvantages.

Experiment setup. Experiments are conducted on a server platform with two ten-core Intel(R) Xeon(R) Gold 5215M CPUs. Hyper-threading is disabled, and the CPU frequencies are set to 2.50GHz to get stable results during the evaluation. The machine is equipped with 384GB DDR4 DRAM and 1.5TB Optane DC persistent memory that is distributed on two NUMA nodes. We perform all experiments on the 750GB Optane DC persistent memory residing on NUMA 0 to ensure stable results.

We evaluate Ext4-DAX [4, 6, 58], PMFS [9], Strata [31], and NOVA [61, 62] and compare the performance of ZoFS to them. We tried to build and run Strata but only succeeded in some benchmarks. We failed to build ZuFS [10] and Aerie [56] using their provided kernel configurations.

6.1 Micro benchmarks

We use FxMark [38] to evaluate the performance of basic operations and stretch the scalability of tested file systems.

Figure 7 shows the throughput of tested file systems as the number of threads increases. ZoFS achieves the best throughput in most workloads, including data reads (Figure 7(a)(b)(c)), data writes (Figure 7(d)(e)(f)) and metadata operations (Figure 7(g)(h)(i)). ZoFS manages data and metadata in user space and prevents context switches and cacheline pollution caused by system calls, which is the major reason for the performance differences.

In the data read workloads, all file systems scale well regardless of the contention level, thanks to the readers-writer lock. For data appends (DWAL, Figure 7(d)), the allocator also affects the performance. ZoFS uses leased per-thread allocator, which scales until 12 threads. After 12 threads the throughput is limited by the `coffer_enlarge` processing in the kernel, which will be further explained later. NOVA uses a per-core allocator, which scales well in this workloads. PMFS, however, uses a global allocator that stops to scale after 4 threads.

Figure 7(e) shows the performance when different threads overwrite the first 4KB block of different files (DWOL), ZoFS hits the write bandwidth ceiling of the NVM we use when the thread number is 12. With the increase of the thread count, the write bandwidth of NVM decreases [25], which reduces the throughput of ZoFS.

Figure 7(f) shows the performance when different threads overwrite different blocks in a shared file (DWOM). Since all tested file systems use per-file locks, the throughput drops as the number of threads increases. ZoFS still presents the best performance under this scenario.

However, when different threads create files in different directories (MWCL, Figure 7(g)), ZoFS stops to scale after 4 threads and is outperformed by NOVA. This is caused by the contention of the `coffer_enlarge` operation. Even if each thread has its private allocator, when it runs out of free space, it will call `coffer_enlarge`. When space allocation is extremely frequent, different threads calling `coffer_enlarge` contend in the kernel and yield non-scalable performance. This is also the reason for the flattened throughput of ZoFS after 12 threads in Figure 7(d). In NOVA, however, each per-core allocator will get an equal share of the whole NVM free space; thus, it will take much longer time to exhaust a per-core allocator in NOVA. So NOVA continues to scale even after 4 threads.

To better illustrate the source of performance gains, we breakdown the throughput of DWOL by modifying the evaluated file systems and show the results in Figure 8. ZoFS-sysempty is a ZoFS variant that issues an empty system call before each file write. ZoFS-kwrite is another ZoFS variant whose file write operation is implemented in kernel space. NOVAi indicates the in-place version of NOVA. The "-noindex" suffix indicates NOVA implementations that do not update indexing structures for file writes. These two "-noindex" NOVA variants are not correct for operations other than file overwrites, and we show them here to illustrate the influence of indexing structure updates. PMFS, by default, uses normal writes followed by `clwb` instructions when `clwb` is available. We force PMFS to use non-temporal writes in the PMFS-nocache variant.

In Figure 8, all systems fall into three groups according to their performance. ZoFS performs the best, followed by ZoFS-sysempty in the fastest group. NOVA-noindex, PMFS-nocache, ZoFS-kwrite, and NOVAi-noindex perform similarly in the second group. PMFS, NOVA, and NOVAi are in the slowest group. The throughput difference between ZoFS and ZoFS-kwrite shows the benefit of implementing file systems in user space. For NOVA implementations, the updates of the indexing structure significantly affect performance. Since all writes in the test are 4KB and aligned, NOVAi has no advantage over NOVA and suffers from its journaled metadata operation. For PMFS, it is surprising that the non-temporal writes are much faster than normal writes followed by `clwb`s. To ensure fairness, we checked the implementation and confirmed that both NOVA and ZoFS use non-temporal writes for all experiments in the paper.

In summary, ZoFS outperforms existing NVM file systems, scales well in almost all workloads, and the user-space implementation contributes to ZoFS's high performance.

Table 6. Filebench workload characteristics.

Workload	# Files	Dir Width	File Size	R/W Ratio
Fileserver	10,000	20	128KB	1:2
Webserver	1,000	20	16KB	10:1
Webproxy	10,000	1,000,000	16KB	5:1
Varmail	1,000	1,000,000	16KB	1:1

6.2 Macro benchmarks

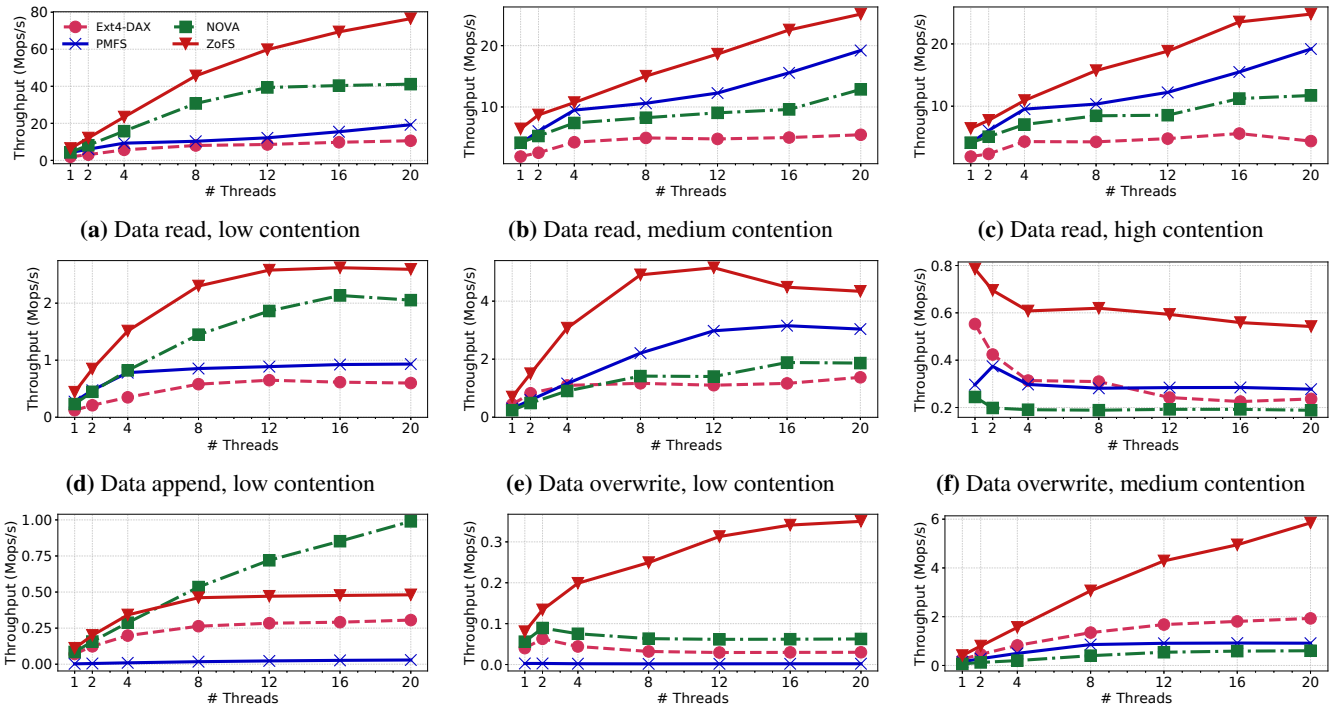
We also evaluate synthesized workloads using Filebench. The characteristics of these workloads are listed in Table 6, and the results are shown in Figure 9.

Generally, ZoFS performs the best in all four workloads. In the single-threaded fileserver workload (also shown in Figure 10(a)), ZoFS outperforms NOVA by 30%, PMFS by 16%, and Strata by 5%. As the thread number increases, ZoFS and NOVA continue scaling, and the performance gap between them shrinks. PMFS scales until 12 threads and falls behind ZoFS and NOVA in throughput. The performance of Strata slightly drops at 2 threads and then stays flat.

In the webserver workload, ZoFS achieves 17% higher throughput than NOVA and 11% higher throughput than PMFS, when there is only one thread. All file systems, except Strata, scale until 12 threads.

In the webproxy and varmail workloads, ZoFS always performs the best. The performance gaps between ZoFS and other file systems enlarge as the number of threads increase. The reason is that the large directory widths (dir-width in Table 6) in the webproxy and varmail workloads cause all files to be stored in a single directory. ZoFS scales well until 12 threads, thanks to its directory design (§5.1). PMFS and NOVA stop to scale earlier, and their performance slightly drops afterwards, due to the inefficiency of finding files in a large directory. To illustrate the impact of the directory width, we also measure the varmail workload with its dir-width set to 20. The result is shown in Figure 10(b). All file systems scale and ZoFS still outperforms PMFS and NOVA by up to 13% and 46%, respectively.

It is worth noting that compared with the performance under the default varmail configuration, the performance of ZoFS drops when the dir-width is reduced to 20. The small dir-width causes deep directory structures. Thus, files accessed in the workload often have long paths. However, ZoFS parses file paths backwards to get the nearest coffer. In other words, starting from the longest prefix, all prefixes of the path are checked until a coffer root is found. This incurs considerable time for files with long paths and slows down the performance of ZoFS. Lines indicated by "ZoFS-20dirwidth" in Figure 9(c) and Figure 9(d) show the performance when the dir-width is set to 20. In comparison with the performance under the default configuration, the throughput of ZoFS decreases by 10% to 30%.



(a) Data read, low contention **(b)** Data read, medium contention **(c)** Data read, high contention

(d) Data append, low contention **(e)** Data overwrite, low contention **(f)** Data overwrite, medium contention

(g) Metadata create, low contention **(h)** Metadata create, medium contention **(i)** Metadata rename, low contention

Figure 7. Results of FxMark workloads. Each data operation accesses files in 4 KB units.

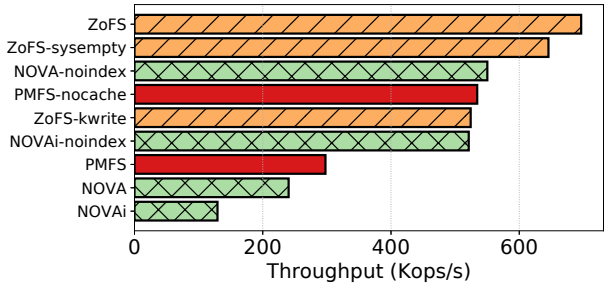


Figure 8. Throughput breakdown of DWOL.

6.3 Real-world applications

In this section, we evaluate the performance of ZoFS and other file systems using two real-world applications: LevelDB and TPC-C SQLite.

LevelDB. LevelDB [14] is a fast key-value storage library that is widely used in cloud environments. We run LevelDB's `db_bench` benchmarks on different file systems and report the results in Table 7. ZoFS has the lowest latency across all file systems and all operations. In some operations such as *write seq.* and *delete rand.*, ZoFS halves the latency compared with PMFS, which presents the second-best latency in the experiment. NOVA adopts the copy-on-write approach and performs worse than PMFS in the test.

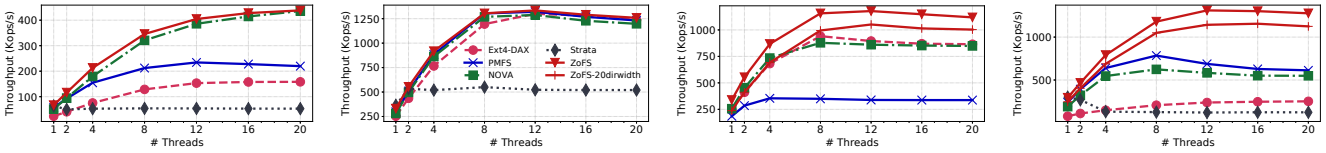
TPC-C SQLite. SQLite [49] is a widely used lightweight yet full-featured SQL database engine. We drive SQLite with TPC-C [53], which is an online transaction processing benchmark that simulates an order processing application.

Table 7. Latency of LevelDB.

Latency/µs	Ext4-DAX	PMFS	NOVA	ZoFS
Write sync.	58.115	23.490	29.055	21.080
Write seq.	7.630	5.019	10.063	3.705
Write rand.	20.052	11.553	19.949	10.296
Overwrite.	30.536	18.223	30.336	16.835
Read seq.	1.389	1.079	1.220	1.071
Read rand.	4.472	3.553	3.990	3.523
Read hot.	1.192	1.164	1.187	1.146
Delete rand.	3.907	2.810	9.418	1.719

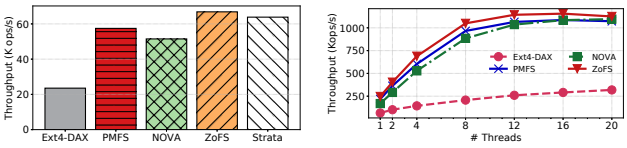
TPC-C involves five types of transactions: New-Order (`NEW`), Payment (`PAY`), Order-Status (`OS`), Delivery (`DLY`), and Stock-Level (`SL`). We use four workloads in the experiment. The first workload is the mixed workload, in which all types of transactions are executed with the ratios given in Table 8. In the other three workloads, we solely execute `NEW`, `OS`, and `PAY` transactions, respectively. We build secondary indexes on the `customer` and `orders` tables and enable foreign keys as required in the specification. We run each workload with a single thread that hosts 1 warehouse and 10 districts.

Figure 11 shows the results. ZoFS achieves the highest throughput in the mixed workload and outperforms PMFS and NOVA by 9% and 31%, respectively. Again, PMFS outperforms NOVA due to the copy-on-write mechanism in NOVA. The performance differences are similar in the other three workloads. The `PAY` workload results in considerably higher



(a) Fileserver **(b)** Webserver **(c)** Webproxy **(d)** Varmail

Figure 9. Filebench.



(a) Fileserver with one thread **(b)** Varmail with dir-width=20

Figure 10. Filebench with customized configurations.

Table 8. TPC-C transaction mix.

Transaction	NEW	PAY	OS	DLY	SL
Ratio	44%	44%	4%	4%	4%

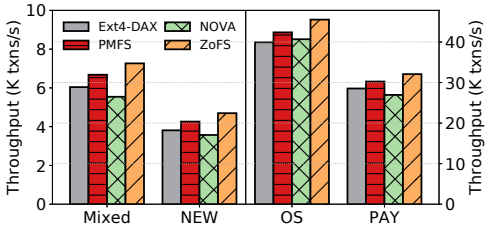


Figure 11. TPC-C SQLite.

Table 9. Worst case performance tests.

Latency/ns	NOVA	ZoFS	ZoFS-1coffer
chmod	1,830	23,342	675
rename	6,261	28,264	1,681

throughput than the NEW workload, because the PAY transaction is much simpler and faster. However, the NEW transactions limit the overall throughput of the mixed workload, although both NEW and PAY account for more than 40% of all transactions. The performance of the OS workload, which is read-only, is even higher than the PAY workload, and ZoFS still outperforms PMFS and NOVA by 7% and 11%, respectively.

6.4 Worst case performance tests

In this subsection, we perform some hand-crafted tests to study how Treasury performs under some worst use cases. Treasury is not good at handling operations across coffers, because these operations must call kernel routines to manipulate coffer's metadata. We write two microbenchmarks, which change the file permission and path, to demonstrate how bad Treasury's performance is in these scenarios. We compare the performance of NOVA, ZoFS, and ZoFS-1coffer (a ZoFS variant that stores all files in one single coffer even if they have different permissions.). Results are shown in Table 9.

In the first microbenchmark, files are stored within one coffer initially. Then we change the permission or ownership of random files. As the results show, ZoFS-1coffer has the lowest latency because it handles all permission updates in user space. NOVA is slower than ZoFS-1coffer because it calls the kernel to change file permissions. ZoFS is about 12× slower than NOVA and 33× slower than ZoFS-1coffer, respectively. This is because it splits its coffer into more and more coffers as the file permissions change. The split procedure will change the coffer of all file pages, which takes a long time. In the other microbenchmark, we have lots of files evenly stored in two coffers[5]. We then randomly pick files and rename them to the other coffer. The result is similar to the chmod microbenchmark for the same reason. Both microbenchmarks confirm the high cost of cross-coffer operations in ZoFS.

6.5 Safety and recovery tests

To verify the protection in the Treasury design, we design some tests, including buggy code and malicious applications to stress the safety of Treasury. We also evaluate the time and effect of recovery in this subsection.

For safety tests, we set up two processes P1 and P2 with two coffers C1 and C2. P1 maps C1 as read-write, and P2 maps both C1 and C2 as read-write.

The first test is about buggy code. P2 regularly accesses files stored in C1 while P1 randomly overwrites its memory space until it is terminated by the kernel. When P1 starts to overwrite outside ZoFS, which simulates the stray writes in application code, file accesses in P2 are never affected since all stray writes in P1 are caught by MPK. When P1 starts to overwrite in ZoFS's code, which can corrupt metadata in C1, P2 receives errors returned by ZoFS during the normal access of files and never terminates unexpectedly. The test presents the effectiveness of MPK protection and graceful error return.

In the second test, P1 acts as an attacker and tries to access the data in C2 by manipulating C1. However, all these accesses are prevented by the kernel. P1 then tries to manipulate C1 to induce P2, which is regularly accessing files in C1, to access files in C2 by mistake. But P2 never accesses files in C2. Instead, it is aware of the manipulated metadata in C1 and reports it as an error. These results show that ZoFS can successfully defend against manipulated metadata from a malicious process.

[5]For ZoFS-1coffer, all files are stored in two directories in a single coffer. For NOVA, the files are stored in two directories.

In the third test, we measure the time and effect of recovering a coffer. When accessing a corrupted file system, the process successfully detected the corruption. We then start the recovery procedure and measure the time. Recovering a coffer with 1,000 2MB-sized files takes 20,748 microseconds in total, with 5,386 microseconds in user space and 15,362 microseconds in the kernel, respectively.

7 Other Related Work

Container-based file systems. The idea of organizing files in a container has been adopted for different purposes in file systems. Chunkfs [18] divides a file system into chunks to improve file system reliability and repair. IceFS [37] introduces cubes, which are used for disentanglement of physical structures. SpanFS [28] proposes domains to improve file system scalability on SSDs. Each domain is a micro file system that manages part of the file system. A similar concept, zoning, is used in BetrFS 0.4 [65] to support fast renaming of full-path indexing. The coffers abstraction in this paper appears as a similar idea. However, it is used to separate protection from management in user-space NVM file systems. Coffers make it possible to directly manage NVM file system data and metadata in user-space libraries while providing sufficient protection and isolation.

User-space file systems and storage management. There are a dozen user-space file systems [15, 16, 34] built on FUSE [51]. These file systems usually suffer from the high overhead of FUSE. ZuFS [10] is a new user-space file system architecture branded with zero-copy. In ZuFS, data will not be copied back-and-forth between user space and kernel space, which significantly reduces the overhead in FUSE. However, file systems using ZuFS still suffer from system calls and context switch overheads that can be prevented in Treasury.

Arrakis [41–43] separates file system naming from file system implementation and lets applications directly manage traditional storage hardware with the help of virtualization. The idea behind Treasury is similar. However, Treasury targets byte-addressable NVM, which is significantly different from traditional storage in both performance and byte-addressability characteristics.

Exokernel. The exokernel architecture separates protection from management to give untrusted applications as much control over resources as possible [2, 11]. Treasury, in some sense, is more like an exo-filesystem architecture, which separates the protection of NVM resources from management and gives user-space libraries full control over NVM. As a result, the untrusted applications can fully exploit the performance of NVM with sufficient protection and isolation.

User-space library protection. Hodor [17] proposes an approach to protecting user-space libraries with Intel MPK. It relies on hardware watchpoints and trust loaders to guarantee safety. ERIM [54] also leverages Intel MPK to provide hardware-enforced in-process isolation and adopts binary inspection to prevent circumvention. Treasury divides the file system into coffers and protects each coffer with the paging mechanism. Treasury adopts Intel MPK and kernel-user cooperation to enforce the isolation and protection of the file system further.

8 Conclusions

This paper introduces a new abstraction, coffers, and a new user-space NVM file system architecture, Treasury. With coffers, Treasury separates NVM protection from management so that NVM performance can be fully exploited with sufficient isolation of NVM resources enforced. Our evaluation shows that an example file system, ZoFS, outperforms existing NVM file systems in both benchmarks and real-world applications.

Acknowledgment

We sincerely thank our shepherd Sam H. Noh and the anonymous reviewers for the constructive comments and suggestions. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104), and the National Natural Science Foundation of China (No. 61772335). Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

References

[1] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 677–694. https://doi.org/10.1145/2983990.2984019

[2] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 43–57. http://dl.acm.org/citation.cfm?id=1855741.1855745

[3] Dave Chinner. 2015. xfs: DAX support. https://lwn.net/Articles/635514/.

[4] Dave Chinner. 2019. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[5] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[6] Jonathan Corbet. 2014. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/.

[7] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/.

[8] Mingkai Dong and Haibo Chen. 2017. Soft Updates Made Simple and Fast on Non-volatile Memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 719–731. http://dl.acm.org/citation.cfm?id=3154690.3154758

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York,

NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[10] Jake Edge. 2018. The ZUFS zero-copy filesystem. https://lwn.net/Articles/756625/.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[12] Julian Gilbey. 2017. Debian Bug report logs - #853972. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=853972.

[13] GNU. 2019. ISO C Thread-local Storage (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/ISO-C-Thread_002dlocal-Storage.html.

[14] Google. 2019. LevelDB. https://github.com/google/leveldb.

[15] Valient Gough. 2019. EncFS: an Encrypted Filesystem for FUSE. https://github.com/vgough/encfs.

[16] GoRed Hat. 2019. Gluster: Storage for you cloud. https://www.gluster.org/.

[17] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process Isolation for High-throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Berkeley, CA, USA, 489–503. http://dl.acm.org/citation.cfm?id=3358807.3358849

[18] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. 2006. Chunkfs: Using Divide-and-conquer to Improve File System Reliability and Repair. In *Proceedings of the 2Nd Conference on Hot Topics in System Dependability - Volume 2 (HOTDEP'06)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=1251014.1251021

[19] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-structured Non-volatile Main Memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 703–717. http://dl.acm.org/citation.cfm?id=3154690.3154757

[20] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.

[21] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-value Stores Using Cross-referencing Logs. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 967–979. http://dl.acm.org/citation.cfm?id=3277355.3277448

[22] Intel. 2018. Intel Optane DC Persistent Memory Readies for Widespread Deployment. https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/.

[23] Intel. 2019. Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[24] Intel. 2019. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm.

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:cs.DC/arXiv:1903.05714v3

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 309–320. http://dl.acm.org/citation.cfm?id=2535461.2535499

[27] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-level Key-value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Berkeley, CA, USA, 191–204. http://dl.acm.org/citation.cfm?id=3323298.3323317

[28] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 249–261. http://dl.acm.org/citation.cfm?id=2813767.2813786

[29] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 993–1005. http://dl.acm.org/citation.cfm?id=3277355.3277450

[30] Takayuki Kawahara. 2011. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *IEEE Des. Test* 28, 1 (Jan. 2011), 52–63. https://doi.org/10.1109/MDT.2010.97

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[32] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[33] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, Berkeley, CA, USA, 73–80. http://dl.acm.org/citation.cfm?id=2591272.2591280

[34] Libfuse. 2019. SSHFS. https://github.com/libfuse/sshfs.

[35] Linux. 2019. dup. http://man7.org/linux/man-pages/man2/dup.2.html.

[36] Linux. 2019. VDSO. http://man7.org/linux/man-pages/man7/vdso.7.html.

[37] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical Disentanglement in a Container-based File System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 81–96. http://dl.acm.org/citation.cfm?id=2685048.2685056

[38] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 71–85. http://dl.acm.org/citation.cfm?id=3026959.3026967

[39] Oracle. 2019. Other MySQL Documentation: Example Databases. https://dev.mysql.com/doc/index-other.html.

[40] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Berkeley, CA, USA, 241–254. http://dl.acm.org/citation.cfm?id=3358807.3358829

[41] Simon Peter and Thomas Anderson. 2013. Arrakis: A Case for the End of the Empire. In *Proceedings of the 14th USENIX Conference on Hot*

Topics in Operating Systems (HotOS'13). USENIX Association, Berkeley, CA, USA, 26–26. http://dl.acm.org/citation.cfm?id=2490483.2490509

[42] Simon Peter, Jialin Li, Doug Woos, Irene Zhang, Dan R. K. Ports, Thomas Anderson, Arvind Krishnamurthy, and Mark Zbikowski. 2014. Towards High-performance Application-level Storage Management. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'14)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=2696578.2696585

[43] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=2685048.2685050

[44] PostgreSQL. 2018. Sample Databases. https://wiki.postgresql.org/wiki/Sample_Databases.

[45] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[46] Ryan Smith. 2015. Intel Announces Optane Storage Brand For 3D XPoint Products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products.

[47] SNIA. 2019. FSL-Dedup Traces. http://iotta.snia.org/traces/5228.

[48] SNIA. 2019. MobiGen Traces. http://iotta.snia.org/traces/5189.

[49] SQLite. 2019. SQLite Home Page. https://www.sqlite.org/index.html.

[50] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *nature* 453, 7191 (2008), 80.

[51] Miklos Szeredi. 2005. Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[52] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 24–24. http://dl.acm.org/citation.cfm?id=2342821.2342845

[53] The Transaction Processing Council. 2019. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[54] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[55] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[56] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

[57] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[58] Matthew Wilcox. 2014. Support ext4 on nv-dimms. http://lwn.net/Articles/588218/.

[59] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 70–83. https://doi.org/10.1145/3173162.3173201

[60] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 349–362. http://dl.acm.org/citation.cfm?id=3154690.3154724

[61] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, USA, 323–338. http://dl.acm.org/citation.cfm?id=2930583.2930608

[62] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[63] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-volatile Main Memories and RDMA-capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Berkeley, CA, USA, 221–234. http://dl.acm.org/citation.cfm?id=3323298.3323319

[64] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 167–181. http://dl.acm.org/citation.cfm?id=2750482.2750495

[65] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2018. The Full Path to Full-path Indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Berkeley, CA, USA, 123–138. http://dl.acm.org/citation.cfm?id=3189759.3189771

[66] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Berkeley, CA, USA, 207–219. http://dl.acm.org/citation.cfm?id=3323298.3323318